The HV-tree: a Memory Hierarchy Aware Version Index

Rui Zhang
University of Melbourne
rui@csse.unimelb.edu.au

Martin Stradling
University of Melbourne
stm@csse.unimelb.edu.au

ABSTRACT

The huge amount of temporal data generated from many important applications call for a highly efficient and scalable version index. The TSB-tree has the potential of large scalability due to its unique feature of progressive migration of data to larger mediums. However, its traditional design optimized for two levels of the memory hierarchy (the main memory and the hard disk) undermines its potential for high efficiency in face of today's advances in hardware, especially CPU/cache speed and memory size. We propose a novel version index structure called the HV-tree. Different from all previous version index structures, the HV-tree has nodes of different sizes, each optimized for a level of the memory hierarchy. As data migrates to different levels of the memory hierarchy, the HV-tree will adjust the node size automatically to exploit the best performance of all levels of the memory hierarchy. Moreover, the HV-tree has a unique chain mechanism to maximally keep recent data in higher levels of the memory hierarchy. As a result, HV-tree is several times faster than the TSB-tree for point queries (query with single key and single time value), and up to 1000 times faster than the TSB-tree for key-range and time-range queries.

1 Introduction

Large enterprises, scientific research and engineering projects commonly require efficient access to temporal databases. For example, a company may keep the changes of salaries of all employees over years. A supermarket may keep the price changes of all products for future analysis. In an astronomy project, each star's position is observed every few days and the position change is recorded over time. In these temporal databases, typical operations are queries by a key, or a time, or both. For example, one may ask: "What is the salary of the CEO?" (by key), or: "Report the salaries of all the employees as of August 2008?" (by time), or: "What was the salary of the CEO as of August 2008?" (by both key and time). Similarly, an astronomer may request for the position changes of a star during the last year. Version indexes provide efficient data access for these types of queries. Various version index structures (e.g., [7, 2, 13, 4]) have been proposed in the last two decades. However, as the data generated from the above applications grows at an unprecedented speed, the existing structures are no longer capable of handling the data in terms of both scalability and efficiency. Wal-Mart's data warehouse was 70 terabytes in 2001 [19], and it became over 1,000 Terabytes in 2007. The Sloan Digital Sky Survey project [1] captures images that cover the sky and the telescope receives 70 gigabytes of images each night. All these new demands call for a version indexing technique that has enormous scalability and extremely high efficiency. In this paper, we design a version index towards this end.

Among many existing version index structures, the TSB-tree [13] has the unique feature of migrating old data progressively to larger and larger mediums; it keeps recent data, which is more frequently accessed, on a fast medium such as the main memory. Therefore, the TSB-tree has the potential of scaling to very large datasets. However, high scalability cannot be achieved without highly efficient tree operations (updates and searches). The TSB-tree's traditional design, which was optimized for two levels of the memory hierarchy (the main memory and the hard disk), undermines its potential for high efficiency in face of today's advances in hardware, especially CPU/cache speed and memory size. As the size of main memory grows rapidly, many databases can now be held in memory, and various studies have shown that improving cache behavior leads to much better performance for memory resident indexes due to growth in CPU/cache speed [15, 16]. The dataset sizes in the aforementioned applications are far beyond the main memory size, but still part of the data will reside in memory. We would like to take advantage of the main memory indexing techniques. Our idea is that the index should optimize cache behavior when data resides in memory and optimize memory/disk behavior when data moves to disk. Moreover, this idea can be extended to any level of the memory hierarchy as data moves along.

In this spirit, we advocate the concept of *full memory hierarchy aware index design*, which tries to optimize performance for all levels of the memory hierarchy rather than just two levels. Guided by this philosophy, we propose a version index called the *memory hierarchy aware version tree* (HV-tree). The HV-tree has nodes of different sizes optimized for different levels of the memory hierarchy. Data will be stored in a node of the best size as data moves to different levels of the memory hierarchy. We summarize the contributions of this paper as follows.

- To our best knowledge, this work is the first attempt for an *index design* that optimizes performance for more than two levels of the memory hierarchy.
- We propose a novel version index structure, the HV-tree, which has different node sizes optimized for different levels of the memory hierarchy. Experiments show that straightforward conversions between nodes of different sizes incur major overhead. A novel design decision here is to change

Proceedings of the VLDB Endowment, Vol. 3, No. 1

the node size *gradually*, which reduces the overhead of node conversions substantially. Moreover, we propose a chain mechanism to maximally keep recent data in higher levels of the memory hierarchy.

- We perform extensive experiments to compare the HV-tree with alternative approaches and the results show that the HV-tree significantly outperforms the competitors in almost all cases. The improvement over the best competitor is up to three orders of magnitude for key-range queries and time-range queries.

The rest of the paper is organized as follows. We first review related work in Section 2. Then we discuss the principles of design for multiple levels of memory hierarchy and two straightforward TSB-tree adaptions towards our goal in Section 3. Section 4 describes the structure and algorithms of the HV-tree. Section 5 reports the experimental results and Section 6 concludes the paper.

2 Preliminaries and Related Work

2.1 Version Indexes

In temporal databases, an object's value may change over time and different values are considered different versions of the object. The database keeps track of the changes of each object's value, and every change is associated with a *version number* or a *timestamp*. Many structures [7, 2, 6, 4, 18, 13] have been proposed to index versioned and temporal data. The most related ones are the Write-once B-tree (WOB-tree) [6], the Multi-version B-tree (MVB-tree) [4] and the Time-Split B-tree (TSB-tree) [13]. They are all variations of the B-tree; they allow nodes to be split by time or by key. A commonality of the WOB-tree and the MVB-tree is that after a time split, the historical data stays in the old node and the current data goes to a new node. The TSB-tree has the unique feature that it puts the old data in the newly created node after a time split and therefore the old data can be *progressively migrated from one storage medium to another* – an important feature that can always keep current data on a high-speed medium (e.g., the main memory). Next, we elaborate the TSB-tree since it serves as the foundation of our HV-tree.

2.2 The TSB-tree

There are a few papers about the TSB-tree [13, 14, 12]. We base our following description on the most recent one [12], which reports a prototype integration of the tree into Microsoft SQL Server.

2.2.1 Structure

Similar to the B$^+$-tree, the TSB-tree is a hierarchical structure consisting of index (non-leaf) nodes and data (leaf) nodes. An index entry is a triple $\langle key, time, pointer\rangle$, where $pointer$ points to a child node N; the values of key and $time$ lower bound the key values and timestamps (respectively) of all the index or data entries in the subtree rooted at N. Each data entry is a triple $\langle key, time, data\rangle$, where key and $time$ indicates the key value and the timestamp of the data value $data$. Entries with the same key value but different $time$ values are considered different versions of the same object. Figure 1 shows an example of a versioned dataset. A black dot/circle represent an insertion/update, respectively. Figure 2 shows the TSB-tree for the dataset assuming a node capacity of 4. At timestamp 3 (Figure 2(a)), there are two versions for key 1, $\langle 1, 1, Alan\rangle$ and $\langle 1, 3, Ben\rangle$. The entry $\langle 1, 1, Alan\rangle$ is *valid* in the time range of $[1, 3)$ and we say its *starting time* and *ending time* are 1 and 3, respectively. Similarly, each index entry (correspondingly each node) in the tree also has a starting time and an ending time.

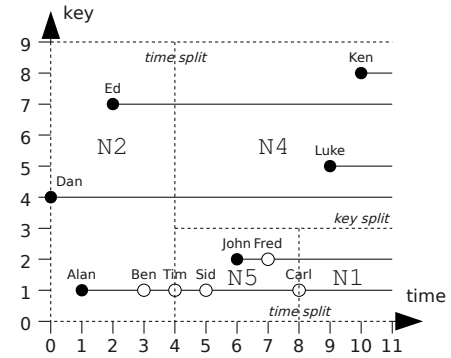


Figure 1: Versioned dataset example.

If an entry/node has an ending time before the current time, we say the entry/node is *historical*; otherwise, we say it is *current*.

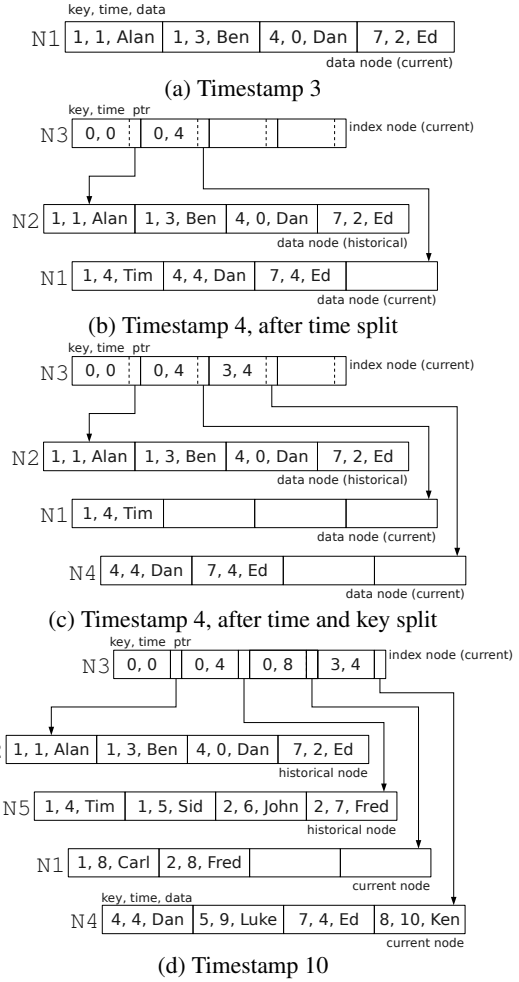


Figure 2: TSB-tree example.

Each index entry (and its corresponding child node) maps to a rectangular range in the key-time space. In Figure 1, the key-time ranges are labelled according to the nodes in Figure 2(d). How to derive the boundaries of the key-time range is explained in Appendix A.1.

2.2.2 Operations

Common index operations are insertion, deletion, update and search. As a versioned index, the TSB-tree keeps all the changes of every object and does not actually delete data. An update is treated as an

insertion with the entry's $time$ value set to the update timestamp. Deletion is a special case of update where a record is flagged as being deleted; then a query beyond the deletion time will ignore the "deleted" record. Therefore, insertion and search are the two main operations of the TSB-tree.

2.2.3 Search and Query Types

The most basic type of query finds records with a given key value and timestamp, which we call a *point query* since it corresponds to a point in the key-time space.

To process a point query with key k_1 and timestamp t_1, the search algorithm traverses down the tree starting from the root node. For a visited node, the algorithm ignores all entries with $time$ value greater than t_1, and finds the entries with the largest key value not exceeding k_1 (there may be multiple entries with the largest key value not exceeding k_1); then among these entries, the algorithm finds the one with the latest $time$ value (, which must not exceed t_1 since entries with $time$ greater than t_1 are ignored). If this entry is an index node entry, the algorithm continues to visit the child node pointed to by this entry and repeat the above process until reaching a data node. Using the same process, the algorithm will find the record we are searching for if the record exists in the database (i.e., the entry found has the key value of k_1 and it is valid at time t_1). An example is given in Appendix A.2.

Besides the point query, we can specify queries with various key and time requirements. A *key-slice time-range query* finds records with a given key value in a given time range. A *time-slice key-range query* finds records in a given key range at a given timestamp. A *key-range time-range query* finds records in a given key range and a given time range, which is the most general form of the queries. A *key-range time-range query* can be processed as follows. Every entry corresponds to a region (or a point in the case of a data node entry) in the key-time space as discussed in Section 2.2.1. Starting from the root node, add to a FIFO stack the nodes whose key-time regions intersect that of the query. Pop the nodes out of the stack one by one and check whether its child nodes intersect the query in the key-time space. Intersected nodes are inserted to the stack. Continue this process until the stack is empty. The records that intersect the query in key-time space are the answers.

2.2.4 Insertion

The insert algorithm first uses the search algorithm to locate the data node to place the new record in. If the data node is not full, the new record is simply inserted into it. If the node is full, it will be split in time space or key space in order to accommodate the new record. The node split generates a new index entry to be added to the parent node, and the split may propagate to upper level nodes if the upper level nodes are also full. Data nodes and index nodes are split differently. We explain the split of data node next and index node split in Appendix A.4.

Data Node Split. The TSB-tree may perform a key split or a time split. A key split on a data node is very similar to a node split in the B^+-tree, where entries with keys smaller than the split value go to one node and the other entries go to another node. When a data node is time split, entries with starting time after or equal to the split time remain in the current node; entries with ending time before the split time go to a new node; entries with time extents spanning the split time are copied to both nodes. As a result, historical entries move to a new node; this new node is a historical node and never needs to update its content again, so it is moved to disk. Current entries remain in the old node in memory.

The decision on whether to split a node by key or by time is made based on a parameter called $SVCU$ (Single-Version Current Utilization), defined as the size of a node's current data divided by the node size. If $SVCU$ is smaller than a threshold T, which means there are many historical entries, then a time split is performed. Otherwise, a key split is performed. In the Microsoft prototype implementation [12], (i) whenever a key split is needed, a time split is always performed before the key split to guarantee a minimum storage utilization for any version, (ii) the suggested value for T is 0.67, and (iii) the splitting time is always the current time. An example is given in Appendix A.3.

2.3 Main Memory Indexing Techniques

Most traditional index structures have been designed to optimize performance for secondary memory such as the B^+-tree and the R-tree [8]. As main memory becomes larger and cheaper, it is feasible to hold large datasets entirely in memory. The performance issue is then how to optimize these structures to reduce the number of cache misses. This problem becomes more important as the gap between the CPU speed and the main memory speed enlarges [5]. Several papers [16, 17, 15] have studied indexing and querying main memory databases. They mainly use the following two techniques to improve cache performance:

(i) Aligning node size with block size. It aligns the node size with the cache block size to reduce cache misses. Both the CSS-tree [15] and CSB^+-tree [16] use the cache block size as the tree node size.

(ii) Pointer elimination. It uses computation to find tree nodes and get rid of pointers, so that more data can fit into a cache block. For example, the CSS-tree arranges its nodes in an array to accelerate search. To handle updates better, the CSB^+-tree keeps some pointers, but the number of pointers is much smaller than a normal B^+-tree. Finding a node is through a pointer to a group of nodes and offsets to member nodes of the group.

2.4 Memory Hierarchy Aware Design

Optimization for multiple levels of memory hierarchy has been studied widely in hardware and algorithm designs (e.g.,[3, 20]), but there has been very little work that considers multiple levels of memory hierarchy in an index design, especially a version index. Most popular indexes (e.g., B-trees and R-trees) are designed for memory/disk performance. The above described main memory indexes are also designed to optimize performance for only two levels of the memory hierarchy, the cache and the main memory.

2.5 Partial Expansions

The technique of "partial expansions" suggests gradually increasing the size of an overflown bucket to increase the space utilization rate (e.g., [10, 11]). In our case, the gradual expansion of nodes has a positive effect on space utilization, but more importantly, it is used to avoid the overhead of directly converting small nodes into a big one.

3 Design for Multiple Levels of Memory Hierarchy

In this section, we discuss our principles towards a version index design that optimizes performance for multiple levels of the memory hierarchy. We also present two straightforward adaptions of the TSB-tree attempting the design goals.

3.1 Design Principles

Memory hierarchy refers to the hierarchical arrangement of storage in a computer, which is designed to take advantage of memory locality in computer systems. The memory hierarchy of a modern computer typically consists of L1 cache, L2 cache, main memory,

hard disk and tertiary storage (such as optical disk and tape). Each level of the hierarchy is mainly parameterized by latency, block size and capacity. Higher levels of the hierarchy usually have smaller block size, capacity as well as smaller latency. The difference of latency between two levels of the hierarchy is up to several orders of magnitude. Therefore, taking full advantage of the memory hierarchy is essential for significant performance improvement. We follow two major Principles in our design:

1. Tailor the index's structure to suits the characteristics of each level of the memory hierarchy.
2. Keep as much as possible frequently accessed data in higher levels of the memory hierarchy.

For all the TSB-tree variants as well as our HV-tree, we assume that main memory is sufficient to hold all current nodes. The TSB-tree naturally complies with Principle 2 since current data, which is more frequently accessed, remains in main memory and historical data, which is less often accessed, is migrated progressively to lower levels of the memory hierarchy. Therefore, the TSB-tree serves as a good starting point for our design. Next, we try to apply also Principle 1 to the TSB-tree.

The main technique for optimizing for a memory level is the simple rule of *aligning node size with block size*. Traditional index designs set the node size to the hard disk page size (typically 4KB or 8KB). This optimizes the main memory and hard disk performance since the data transfer unit between the main memory and hard disk is the hard disk page size (and hence data is stored as 4KB or 8KB blocks both in main memory and on hard disk). The TSB-tree already does this, so our research focuses on how to optimize the TSB-tree for cache behavior (we will discuss memory levels beyond hard disk in Section 4.3.2).

Similarly, the trick to optimize for cache behavior is to set the node size to be the cache block size, and this trick has been used by a few previous proposals as discussed in Section 2.3. However, a later study [9] shows that the actual optimal node sizes for the CSB$^+$-tree [16] are much larger than the cache block size due to the number of instructions executed and the number of TLB misses. The method in [9] cannot be directly applied to determine the optimal node size for the TSB-tree, but it warns us to pick the optimal node size with caution. Let S_{cache} denote the optimal node size for the cache performance of the TSB-tree. Determination of S_{cache} is highly hardware dependent and a through study on it is beyond the scope of this paper. In this study, we take an empirical approach and just run some experiments to find S_{cache}, which turns out to be 1KB in our case. Details are given in Appendix D.1. In the rest of the paper, we assume S_{cache} is a known parameter. Please note that finding S_{cache} optimizes for the combined effect of all the caches, i.e., L1, L2 and L3 (if present) caches.

3.2 Straightforward TSB-tree Adaptions

Following the above discussion, we provide two straightforward TSB-tree adaptions attempting to optimize cache performance as follows.

The first TSB-tree adaption simply uses S_{cache} as the node size. The idea is to optimize the tree for cache behavior and let the buffer manager of the operating system (OS) to deal with paging. We call this TSB-tree variant **TSB-small**. Our experiments show that TSB-small actually performs worse than the TSB-tree. This is because the performance loss caused by bad disk behavior overweighs the performance gain caused by better cache behavior.

The second TSB-tree adaption uses S_{cache} as the current node size and the hard disk page size (denoted by S_{disk}) as the historical node size. We need to somehow convert the node size from S_{cache} to S_{disk} as current nodes become historical and move to the hard disk. S_{disk} is typically a power of 2 times S_{cache} (in our experimental setting, S_{disk} and S_{cache} are 4KB and 1KB, respectively). A straightforward way of converting the nodes is as follows. For the first current node that becomes full and performs a time split, we migrate the resultant historical node to the hard disk and simply expand this historical node's size to S_{disk}. Subsequently, when a time split is performed on a node `N`, instead of immediately creating a new node (for historical entries), we look for a sibling historical node of `N` on the hard disk which has enough space to hold the historical entries (since we will have historical nodes with lots of unused space from the previous size expansion). If we find such a node, then we can put the historical entries of `N` in it. Otherwise, we create a new historical node on the hard disk with size S_{disk} to contain the historical entries of `N`. Details of the node conversion process is given in Appendix C. This TSB-tree variant is characterized by condensing small nodes to large nodes, so we call it **TSB-cond**. Our experiments again show that TSB-cond performs worse than the TSB-tree. This is because node condensation incurs the following two kinds of major overhead:

1. Searching for a sibling and condensing the nodes cause many extra I/Os (historical nodes are on disk) and computation.
2. The condensation results in bad clustering and low utilization rate in historical nodes. In many cases, an index node is time split earlier than many of its child data nodes are. After an index node time split, the resultant historical index node will never gets updated and its child nodes will never gets condensed, so the child nodes remain highly underutilized.

4 The HV-tree

In this section, we present the structure and algorithms of the HV-tree. An important and novel design of HV-tree is that its node size changes *gradually* rather than directly from S_{cache} to S_{disk}. This design makes it possible to have different node sizes without the overhead to convert between them compared to TSB-cond. Moreover, we propose a chain mechanism to keep the maximum amount of recent data in memory to further improve the performance (this can be viewed as an enhancement of Principle 2). Table 1 summarizes the symbols used.

Name	Description
T	Splitting threshold value
S_{cache}	Optimal node size for cache performance
S_{disk}	Optimal node size for memory/disk performance
$F_t(s)$	Node fanout for node type t and node size s
R_c	Number of current entries in a node
O_{mig}	Offset for migrated data
O_{hist}	Offset for historical data

Table 1: Symbols

4.1 Structure

The structure of the HV-tree is similar to that of the TSB-tree, but the HV-tree has nodes of different sizes. The size of an HV-tree node may be S_{cache}, $2S_{cache}$, ..., S_{disk} (note that S_{disk} is a power of 2 times S_{cache}). For example, in our experimental settings, S_{cache} is 1KB and S_{disk} is 4KB, so an HV-tree node may be of the size of 1KB, 2KB or 4KB.

Compared to a data node of the TSB-tree, a data node of the HV-tree has an additional pointer, $nextOldest$, in the header information; other parts of an HV-tree data node is the same as a TSB-tree data node. The additional pointer is used in the data migration process as explained in Section 4.3.

An index node of the HV-tree has two differences from that of the TSB-tree.

1. An HV-tree index node has an additional pointer, $nextOldest$, in the header information and it is used in data migration.
2. An HV-tree index entry is of the form $\langle key, time, pointer, histPtr \rangle$; it has an extra pointer, $histPtr$, in addition to a TSB-tree index entry. The first three fields, key, $time$, and $pointer$, are used in the same way as in the TSB-tree; $histPtr$ is used in data migration.

4.2 Insertion

We call S_{cache}, $2S_{cache}$, ..., S_{disk} the *allowable sizes* of an HV-tree node. In the HV-tree, we always try to give the smallest allowable size to a current node when possible. The rationale is to make their sizes close to S_{cache} to optimize cache behavior. When start building an HV-tree, the first data node is given the size of S_{cache}. When the root is split and we create a new root for the tree, the new root is also given the size of S_{cache} (the new root has only two entries, so it can always fit into S_{cache} assuming that S_{cache} can contain at least two entries). In all other cases, we grow the tree using the split procedure described as follows.

4.2.1 Splitting

When an entry is inserted into a full node, the HV-tree may perform a time split, a key split or a *node expansion*. Node expansion is a new operation compared to those of the TSB-tree. The decision on which action to perform depends on the portion of current entries in the node. When the portion of current entries is greater than or equal to a threshold T, then it is worth performing a key split, resulting in two current nodes. The the two resultant current nodes can fit into smaller allowable node sizes; we will give each resultant current node the smallest allowable node size that can contain the corresponding current entries.

When a node is full and the portion of current entries is smaller than T, if the node size is smaller than S_{disk}, we double the size of the node – this is a critical difference from the TSB-tree; otherwise (the node size is already S_{disk}), we perform a time split. After the time split, only current entries remain in the original node. Since the number of entries in the original node has decreased, we contract the size of the original node to be the smallest allowable size that can contain all the current entries. All the historical entries are put in a new node of size S_{disk} and the new node is added to the tail of the migration chain. The historical nodes will be actually moved to disk when we run out of memory. We will explain the node migration process in Section 4.3.

The above steps are summarized in Algorithm 1. Note that this algorithm applies for both data and index nodes. The input of the algorithm is a node N and its $type$ (`index node` or `data node`). The function GetCurrentNodeNumber(N) returns the number of current entries in node N (line 1). The function $F_{type}(N.\text{header.size})$ returns the fanout of the node based on the node type and the current size of the node which is given in the header information. Then we can compute the portion of current entries β. The flow chart in Figure 3 helps understanding the HV-tree splitting process and a running example is given in Appendix B. One may ask: can we check the condition of $\{nodesize$ vs. $S_{disk}\}$ before checking the condition of $\{\beta$ vs. $T\}$? The answer is "No". If we do so, we will always expand a node until the node size reaches S_{disk}, and then we start to choose between key split and time split. Then it is no different from the original TSB-tree splitting procedure.

Choice of T. T is an important parameter. It controls what types of actions are performed upon inserting into a full node and also affects node utilization. If T is large, there are more time splits and hence more data redundancy. It may result in better query performance for time-slice queries, but worse for other types of queries. If T is small, there are more key splits and less data redundancy, which favors overall query performance.

Algorithm 1: splitNode(N, *type*)

```
1  R_c ← GetCurrentNodeNumber(N)
2  β ← R_c / F_type(N.header.size)
3  if β ≥ T then
4      KeySplitNode_type(N)
5  else
6      if N.header.size < S_disk then
7          ExpandNode(N)
8      else
9          TimeSplitNode_type(N)
10         ContractNode(N)
11         add historical node to migration chain
12 if out of memory then
13     Perform data migration
```

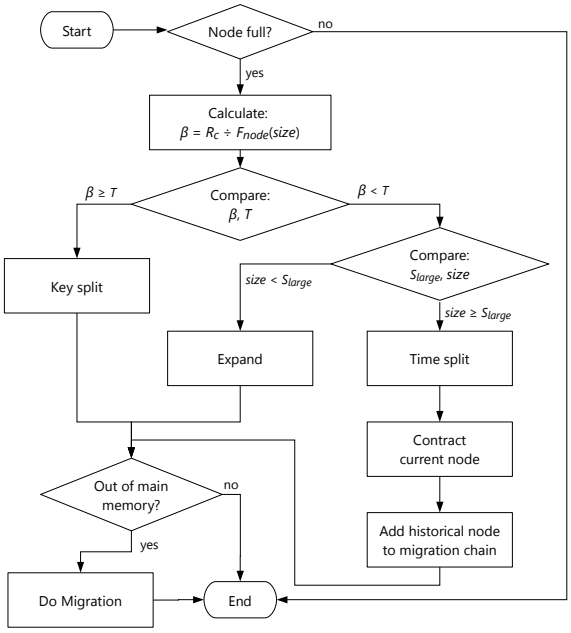


Figure 3: HV-tree splitting policy.

We observe that when a node is time split, its size is S_{disk}. After the time split, less than T portion of the entries remains in the current node, and more than $(1 - T)$ portion of the entries move to a historical node. We would like the current node to fit into the smallest allowable node size for optimizing cache performance, so T should be smaller than $\frac{S_{cache}}{S_{disk}}$. At the same time, a small T value results in a large value of $(1-T)$ and therefore high utilization rate in historical nodes. However, setting T to be too small will cause too many key splits and result in a very low single version utilization rate (SVCU), which significantly hurts the time-slice query performance. Based on the above analysis, we choose the value of T to be $\frac{S_{cache}}{S_{disk}}$. In our experimental settings, S_{cache} = 1KB and S_{disk} = 4KB; therefore T is 0.25. This value of T guarantees a high utilization rate of 75% for historical nodes. Our experiments also validates that it is a very good choice.

4.3 Data Migration

In the HV-tree, historical nodes are not moved to the hard disk immediately after creation. Instead, they are linked through a migration chain and when the memory runs out, they will be moved to the hard disk in the order of their creation time. This technique allows us to keep as many as possible the most recent historical nodes in the memory. In what follows, we describe first the chain mechanism that links the historical nodes and then the process of progressive migration of the nodes to lower memory levels.

4.3.1 Migration Chain

The HV-tree maintains two pointers, $firstOldest$ and $lastOldest$, to point to the start and the end of the migration chain, respectively, as shown in Figure 4. When the first historical node is created,

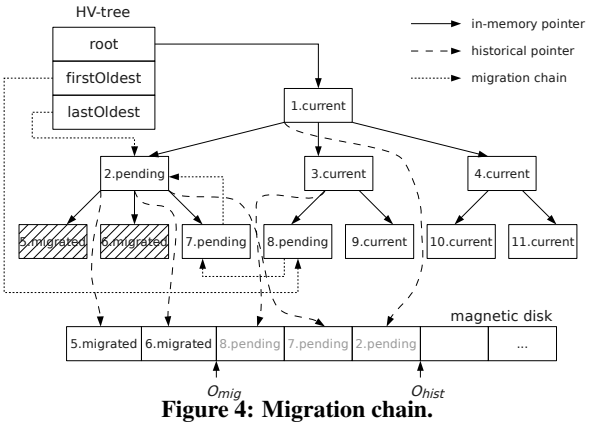


Figure 4: Migration chain.

both pointers point to that node. Then the chain is maintained as follows. Suppose currently $lastOldest$ points to $N1$ and then a new historical node N gets created. We first let $N1$'s $nextOldest$ point to N and then let $lastOldest$ point to N. Thus we have a chain of historical nodes in the order of their creation time.

Let p denote the index entry that points to N when N is created. Another thing we do upon creation of a historical node N is to assign a hard disk address for the future migration of N and set $p.histPtr$ to this hard disk address. The historical nodes are migrated to the hard disk in the order of the migration chain: the first node migrated to the first block on the hard disk, the second node migrated to the second block on the hard disk, and so on. We use a variable O_{hist} to indicate where a newly created historical node will be placed on the hard disk and O_{hist} is initialized to 0. When a historical node N is created, the value of O_{hist} is given to $p.histPtr$ and then incremented by S_{disk}.

The reason we set the value of $p.histPtr$ upon N's creation rather than at a later time (say, the time of the actual migration) is as follows. Node N's parent may be time split and the two resultant nodes (one current and one historical) may both have pointers to N. If we set the value of $p.histPtr$ later, we have to pay the cost of finding both parents of N, and more importantly, we need to update the historical parent of N, which is not supposed to be updated anymore.

Since we need to set the value of $p.histPtr$ while $p.pointer$ is valid, we must keep both pointers in an index entry.

The historical nodes are migrated to the hard disk in the order of their creation time. Therefore, the nodes remaining in the main memory are always the most recent ones. We maintain another variable O_{mig} to indicate where the next migrated historical node should be on the hard disk and O_{mig} is initialized to 0. When a historical node is migrated to the hard disk, O_{mig} is incremented by S_{disk}. Another use of O_{mig} is to help determine whether a historical node has been migrated to the hard disk or not when searching for the historical node, as explained in subsequent subsections.

Figure 4 shows an example of the HV-tree's migration chain. An arrow with a solid line represents a pointer denoted by the $pointer$ field in an index entry. An arrow with a dashed line represents a $histPtr$ pointer. Arrows with dotted lines represent the migration chain. The texts inside the nodes indicate the nodes' status. For example, nodes 5 and 6 have been migrated to disk, so the memory they once occupied can be used by other nodes. Nodes 8, 7 and 2 are part of the migration chain and will be migrated next if needed.

4.3.2 Migration Beyond Hard Disk

We still want to keep the most recent data in higher levels of the memory hierarchy when migrating data to even lower memory levels. Suppose there are n memory levels beyond the main memory, L_1, L_2, ..., L_n, and the capacity of L_i is C_i, where $i = 1, 2, ..., n$. For example, L_1 may be the hard disk and L_2 may be a tape.

We need to be able to perform two operations. The first one is how to *migrate a node* down the hierarchy, and the second one is how to *locate a node* in the memory hierarchy when retrieving it.

Migrate a node: When we want to migrate a node N from the main memory to L_1, we check whether $O_{mig} \leq C_1$. If yes, we write N to location O_{mig} of L_1. If not, it means that we have filled C_1 entirely. To keep recent historical nodes on higher memory levels, we put N at location $(O_{mig} \bmod C_1)$ of L_1 and move the existing node at location $(O_{mig} \bmod C_1)$ of L_1 to location $((O_{mig} - C_1) \bmod C_2)$ of L_2. Figure 5 gives an example where we have three memory levels with the capacities of 2, 4 and 8 blocks, respectively. A number i in a block denotes the i^{th} created historical node. When we try to migrate the third historical node, L_1 is full and therefore we first move the node at location $(O_{mig} \bmod C_1) = (3 \bmod 2) = 1$ of L_1 to location $((O_{mig} - C_1) \bmod C_2) = ((3 - 2) \bmod 4) = 1$ of L_2. Then we can migrate the third historical node to location 1 of L_1. In a similar manner, we migrate the fourth, fifth and sixth historical nodes. When we try to migrate the seventh historical node, we need to first migrate the fifth historical node to location 1 of L_2, but location 1 of L_2 is occupied by the first historical node. In this case, we need to first migrate the first historical node to L_3, then migrate the fifth historical node to L_2, and finally migrate the seventh historical node to L_1. The formula we use to move the first historical node from L_2 to L_3 is similar to the one we use to move it from L_1 to L_2.

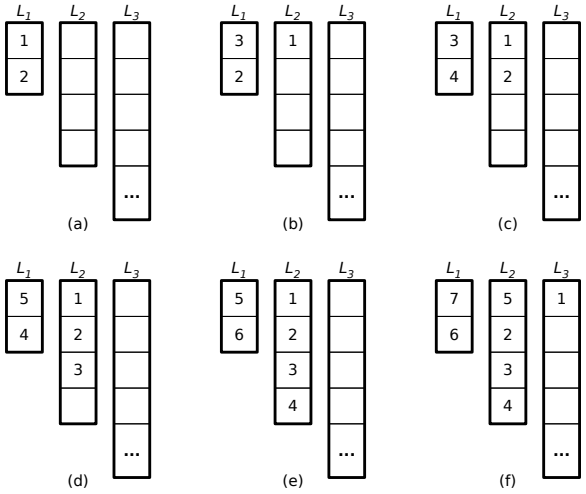


Figure 5: Data migration beyond hard disk.

Locate a node: Suppose we want to locate node N and its parent index entry is p. If $p.histPtr$ is beyond O_{mig}, it means that N is still in memory and we follow $p.pointer$ to locate N. Otherwise N has been migrated and its address is given by $p.histPtr$. Given $p.histPtr$, we can use the reverse address computation of what is used in data migration to find the location of N.

4.4 Search

Searching in the HV-tree follows the same procedure as the TSB-tree except that we will follow $pointer$ of an index entry only when $histPtr$ is larger than O_{mig} (which means the node pointed by $pointer$ has not been migrated). Otherwise, we will follow $histPtr$ to locate the node.

4.5 Summary

Compared to the TSB-tree, the HV-tree has better cache behavior for current nodes, better space utilization based on our analysis on T, and better memory caching since it always keeps maximally the most recent historical nodes in memory. Table 2 summarizes the major differences between the HV-tree and the TSB-tree.

HV-tree	**TSB-tree**
Variable node sizes	Fixed node sizes
A full node can be key split, time split or expanded	A full node can be key and time split, or just time split
Delays migration of data	Immediately migrates data
Index node splits depend on R_c	Index nodes are always time split if possible

Table 2: HV-tree vs. TSB-tree.

Compared to TSB-cond, the HV-tree avoids node condensation and therefore does not have the problems of extra I/Os and low space utilization rates.

5 Experiments

In this section we will compare the HV-tree with the standard TSB-tree (denoted as "TSB-stand"), TSB-small and TSB-cond as described in Section 3.2. Except the HV-tree, all the TSB-tree variants are allowed to use the buffer manager of the operating systems to cache the pages. The test parameters and their default values are described as follows.

We denote the percentage of updates among the total number of insertions and updates by u, and we call this parameter the update frequency. The default value for u is 80%. We assume that recent data are more frequently queried, so query time follows a Zipfian distribution skewed towards the current time, with a default α value of 1. The search keys are randomly chosen from the existing keys. We measure the total response time and the number of disk I/Os. For all the query types, we run 500 queries following the above described distribution and report the average result.

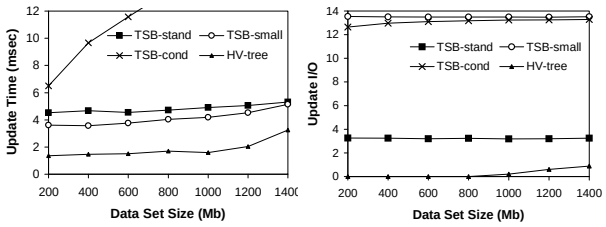
We ran all the experiments on an PC with an Intel P4 3GHz CPU, 1GB of memory and an 80GB hard drive. The hard disk page size is 4KB. The capacity, block size, and associativity are $\langle$8KB, 64B, 1$\rangle$ for the L1 cache and $\langle$512KB, 64B, 8$\rangle$ for the L2 cache.

We vary the dataset size from 200MB up to 1,400MB so that we can observe the behavior of the various techniques when the dataset size is smaller and larger than the memory size. Every 500MB of data corresponds to about 40,000,000 records where each record has 4 bytes of data besides its key and time.

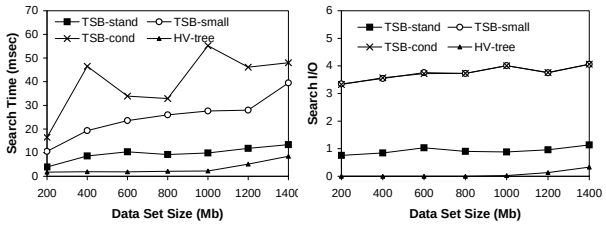
We focus on presenting the representative results, while additional results are given in Appendix D. For our HV-tree, we find S_{cache} to be 1KB through experiments (Appendix D.1); S_{disk} is the hard disk page size 4KB.

5.1 Updates and Point Queries

We perform 500 point queries with keys randomly distributed and time following Zipfian distribution as described above. We vary the dataset size from 200MB to 1,400MB. Figure 6(a) shows the update performance results and Figure 6(b) shows the search performance results.



(a) Update



(b) Search

Figure 6: Updates and Point queries, varying dataset size.

Figure 6(a) shows that the HV-tree has superior update performance over all TSB-tree variants, in both time and I/O. TSB-small and TSB-cond are worse than TSB-stand in most cases. The HV-tree only accesses the hard disk for the larger datasets, while TSB-stand accesses the hard disk for all sizes and have significantly more disk I/Os than the HV-tree for the largest dataset. For the largest dataset (1,400MB), the HV-tree is 4 times faster than the best TSB-tree variant. The superior update performance is due to the HV-tree's better cache behavior and optimized data migration.

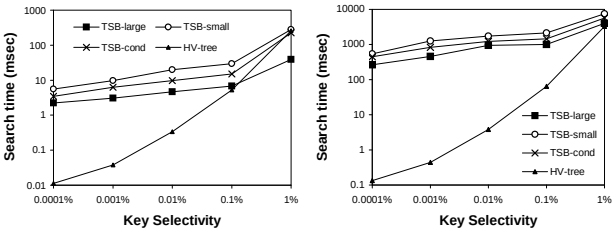
When searching for point queries (Figure 6(b)), TSB-small performs worse than TSB-stand due to bad disk performance. TSB-cond has the worst performance because of the extra I/Os caused by searching for a sibling during node condensation and low utilization rates caused by node condensation. The HV-tree clearly has the best performance in both time and I/O. The HV-tree uses no disk I/O until main memory is exhausted. It is over 6 times faster than the best competitor, TSB-stand, before the dataset size reaches 1GB and still 30% faster than TSB-stand after the dataset size goes beyond 1GB. Please note that the search operation does not involve any data migration. The better performance is mainly due to the better cache behavior of the current nodes of the tree, better historical node utilization and more recent nodes kept in the main memory compared to TSB-stand, which uses the OS buffer to cache historical nodes.

We have run experiments with varied update frequency and skewness of the data (results in Appendix D.2 and D.3). The HV-tree always outperforms the other techniques. In summary, the HV-tree is the best performing method in terms of update and search.

5.2 Key-Range Queries

Figure 7(a) shows time-slice queries as we vary key selectivity from 0.00001% to 1% on the standard 500MB dataset. We see that for selectivities less than 0.1%, the HV-tree is more than two orders of magnitude faster than all TSB-tree variants. The improvement of the HV-tree over other techniques is much higher than in update/search operations because the key-range query involves re-

turning a lot more records. At 0.1% the HV-tree and TSB-stand are roughly equal and beyond this at 1% selectivity, TSB-stand performs the best. This is because TSB-stand's policy of always performing a time split before a key split results in a tree with very high single version utilization rate, which strongly favors the time-slice query especially when the selectivity is large. However, this is a trade-off the HV-tree makes for better performance in all other query types. Even for the time-slice query, the HV-tree is far better than TSB-stand in small selectivities.



(a) Time-slice (b) Time-range

Figure 7: Key-range queries with 500MB dataset.

Figure 7(b) uses the same key selectivities over the same dataset but this time we perform queries which have a time range also. The default time selectivity we chose was 50%, i.e., on average we query half the history of all records in the key range. We see a similar trend as in the time-slice query test. The HV-tree is up to three orders of magnitude faster for the lowest selectivities. Even for the highest selectivity 1%, all methods perform similarly. This is because we are performing a very large query and the volume of data to be returned is the major factor in the cost of the queries.

We repeated the previous tests with 1,000MB datasets and the results show similar behavior. Details are given in Appendix D.4.

5.3 Space Utilization

We expect the HV-tree to have better space utilization than TSB-tree variants. This is because there is less redundancy due to less time splits. However, the fanout will be lower for the HV-tree because it uses smaller nodes and each entry in an index node is larger due to the additional $histPtr$. We compare the sizes of the HV-tree and all the TSB-tree variants for two 480MB datasets, with update frequencies (u) of 25% and 75%, respectively. Table 3 shows the results.

u	HV-tree	TSB-stand	TSB-cond	TSB-small
25%	524.40MB	964.57MB	970.58MB	965.43MB
75%	512.08MB	807.86MB	834.41MB	834.35MB

Table 3: Space utilization for 480MB dataset.

The HV-tree is the smallest among all the structures for both datasets. Compared with the best TSB-tree variant (TSB-stand), the HV-tree uses 45% and 36% less space for the 25% and 75% update frequency datasets, respectively. This verifies that the HV-tree has better space utilization and the additional pointers and smaller nodes do not cause space utilization problems.

5.4 Other Experiments

We have performed experiments on time-range queries using both 500MB and 1,000MB datasets. The HV-tree outperforms other TSB-tree variants by about 1000 times in all cases. Details are given in Appendix D.5.

We also performed a test to see the effect of T on the performance of the HV-tree. The results validate that our analysis provides a good choice for the value of T. Details are given in Appendix D.6.

6 Conclusions

We have presented the HV-tree, a highly efficient and scalable version index. The HV-tree allows different nodes sizes to optimize for different levels of the memory hierarchy. It also has mechanisms to smoothly convert the different sizes of nodes. For almost all types of queries, the HV-tree is significantly more efficient than TSB-trees. The improvement is up to 1000 times for key-range and time-range queries. The HV-tree is the first index design that effectively uses the caches, main memory, hard disk and further memory levels as a whole. We have provided analysis on how to choose important parameters for the HV-tree and the analysis was also justified by experiments.

As future work, it is possible to design the HV-tree to exploit multi-core CPUs for further performance improvement.

7 Acknowledgments

This work is supported by the Australian Research Council's Discovery funding scheme (project number DP0880250).

8 References

[1] *http://www.sdss.org/*.

[2] C.-H. Ang and K.-P. Tan. The interval b-tree. *Inf. Process. Lett.*, 53(2):85–89, 1995.

[3] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *MICRO*, pages 245–257, 2000.

[4] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *VLDB Journal*, 5(4):264–275, 1996.

[5] P. A. Bernstein, M. L. Brodie, S. Ceri, D. J. DeWitt, M. J. Franklin, H. Garcia-Molina, J. Gray, G. Held, J. M. Hellerstein, H. V. Jagadish, M. Lesk, D. Maier, J. F. Naughton, H. Pirahesh, M. Stonebraker, and J. D. Ullman. The asilomar report on database research. *SIGMOD Record*, 27(4):74–80, 1998.

[6] M. C. Easton. Key-sequence data sets on inedible storage. *IBM Journal of Research and Development*, 30(3):230–241, 1986.

[7] R. Elmasri, G. T. J. Wuu, and V. Kouramajian. The time index and the monotonic b+-tree. In *Temporal Databases*, pages 433–456. 1993.

[8] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[9] R. A. Hankins and J. M. Patel. Effect of node size on the performance of cache-conscious b+-trees. In *SIGMETRICS*, pages 283–294. ACM, 2003.

[10] P.-Å. Larson. Performance analysis of linear hashing with partial expansions. *TODS*, 7(4):566–587, 1982.

[11] D. B. Lomet. Partial expansions for file organizations with an index. *TODS*, 12(1):65–84, 1987.

[12] D. B. Lomet, M. Hong, R. V. Nehme, and R. Zhang. Transaction time indexing with version compression. *PVLDB*, 1(1), 2008.

[13] D. B. Lomet and B. Salzberg. Access methods for multiversion data. In *SIGMOD Conference*, pages 315–324, 1989.

[14] D. B. Lomet and B. Salzberg. The performance of a multiversion access method. In *SIGMOD Conference*, pages 353–363, 1990.

[15] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *VLDB*, 1999.

[16] J. Rao and K. A. Ross. Making b$^+$-trees cache conscious in main memory. In *SIGMOD Conference*, 2000.

[17] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *VLDB*, 1994.

[18] Y. Tao and D. Papadias. Mv3r-tree: A spatio-temporal access method for timestamp and interval queries. In *VLDB*, pages 431–440, 2001.

[19] P. Westerman. *Data warehousing: using the Wal-Mart model*. Morgan Kaufmann, 2001.

[20] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. *SIGPLAN Not.*, 26(6):30–44, 1991.

APPENDIX

A TSB-tree Descriptions

A.1 Determination of Key-time Range of an Index Entry

The lower bound is already known from the index entry itself. We use entry $\langle 0, 4, \star$N5$\rangle$ in the root node N3 (Figure 2) as an example to show how to determine the upper bound of the key-time range of an index entry. For the time upper bound, we search in N3 for an entry with the same key (0) and with the smallest time greater than 4, which is $\langle 0, 8, \star$N1$\rangle$, so the time upper bound is 8. For the key upper bound, we search in N3 for an entry with key greater than 0, and with time equal to or less than 4. $\langle 3, 4, \star$N4$\rangle$ satisfies the condition, so the key upper bound is 3. If no entry satisfies the above conditions, then the time upper bound is the current time and the key upper bound is infinity.

A.2 Example of Search for Point Queries

For example, suppose the current timestamp is 10 and we are searching for the record with the key value of 1 at timestamp 6. The TSB-tree at timestamp 10 is shown in Figure 2 (d). Starting from the root N3, we ignore all entries with times greater than 6 (i.e., entry $\langle 0, 8, \star$N1$\rangle$), and find the largest key not greater than 1 (i.e., key 0). The latest entry with key 0 has time 4. We follow the pointer to the child node N5. In N5, we again ignore all entries with time values greater than 6 (i.e., $\langle 2, 7, Fred\rangle$), and find the largest key not greater than 1 (i.e., key 1). The latest entry with key 1 is $\langle 1, 5, Sid\rangle$ and we get the answer, Sid.

A.3 Example of Data Node Split

For the example in Figure 2, the initial insertion of key 4 at time 0, key 1 at time 1, key 7 at time 2, and an update on key 1 at time 3 results in a full data node N1 as shown in Figure 2(a). The update on key 1 at time 4 causes N1 to be split. We calculate $SVCU$ to be $3/4 = 0.75$ (3 current entries and 4 entries), greater than T = 0.67, so N1 will be time split and then key split. First, it is split at the current time 4. Entries $\langle 1, 1, Alan\rangle$ and $\langle 1, 3, Ben\rangle$ have ending times earlier than 4 and are moved to a new node N2 (historical node). The new entry, $\langle 1, 4, Tim\rangle$ has a starting time equal to the split time and therefore is put in the current node. The other entries $\langle 4, 0, Dan\rangle$ and $\langle 7, 2, Ed\rangle$ have time ranges crossing the split time and they are copied to both the new node and the current node; their starting times in the current node N1 are changed to 4 since their time ranges are lower bounded by 4. An index node N3 is created as the new root and two entries are added to it, $\langle 0, 0, \star$N2$\rangle$ and $\langle 0, 4, \star$N1$\rangle$. At this point, the tree is as shown in Figure 2(b). Next, the current node N1 is key split by the value 3, resulting in two current nodes (N1 and N4) and an index entry $\langle 3, 4, \star$N4$\rangle$ added to the root N3. The tree after the update on key 1 at time 4 is shown in Figure 2(c).

The new node N2 only contains historical entries and hence never needs to be updated again. It is then migrated to a lower level of the memory hierarchy (even a write-once medium). This assignment of nodes makes it possible to progressively move historical data to another storage medium, one node at a time.

After time 4, keys continue to be inserted and updated at times 5, 6 and 7. At time 8 we want to update key 1 in node N4 but it is full, so it needs to be split. The $SVCU$ of N4 is calculated to be 0.5, which is less than T, so only a time split is performed. Two more insertions are made at times 9 and 10 with out any overflow and the final tree is shown in Figure 2(d).

A.4 Index Node Split

Index node splitting policies are different from those of data node splitting. Complications arise when splitting index nodes. An index node entry corresponds to a rectangular region in the key-time space. Entries spanning over a key range means that they may need be duplicated in a split. This is not a problem if the entry is historical because it need not be updated again. If the entry is current, duplicating it means that its child node will have two updatable parents; this is a problem because updates on the child node may require updating both parents, but we only know one parent from the search for the child node and there is no easy way to locate the other parent. To avoid this problem, we search for an earlier time which allows the node to be time split; if no such time exists, then a key split is always possible. Splitting at an earlier time reduces the node usage for historical nodes and a key split reduces the fanout for time-slice queries.

B Running Example of HV-tree Splitting

We will use the example in Figure 8 to explain the construction of an HV-tree. In the figure, solid circles represent insertions, hollow circles represent updates and dashed lines represent key/time regions.

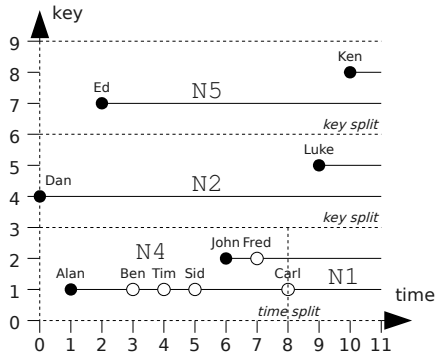


Figure 8: Example data for HV-tree.

In this example, we assume that $S_{disk} = 2S_{cache}$ and therefore the allowable node sizes for the HV-tree in this example is S_{cache} and S_{disk}. Further, we assume that a node of size S_{cache} has a fanout of 3 (for both index and data nodes) and a node of size S_{disk} has a fanout of 6 (for both index and data nodes). We set T to 0.4.

Initially the HV-tree is empty containing only a single data node. Insertion into non-full nodes in the HV-tree is the same as in the TSB-tree. From time 0 to 2, we insert new entries into the only node until it becomes full. The HV-tree after inserting the first three entries is shown in Figure 9.

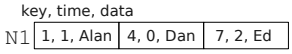


Figure 9: Before a key split.

Key split: At time 3 we attempt to update the entry with key 1, but the node is full so we calculate the R_c of N1. The R_c for a data node is simply the number of current entries it contains, and in this case R_c is 3. R_c divided by the fanout (3/3) is greater than T (0.4). Therefore we perform a key split on the node. A key split in the HV-tree is done in the same as in the TSB-tree. The resultant HV-tree is shown in Figure 10.

Node expansion: The entry with key 1 is again updated at time 4 and 5. At time 5, the data node becomes full. The HV-tree at this moment is shown in Figure 11.

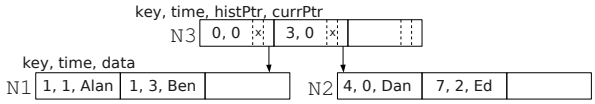


Figure 10: After a key split.

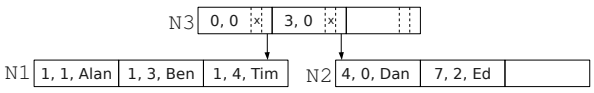


Figure 11: Before a node expansion.

The R_c of the full node N1 is now 1 (there is only 1 current entry). R_c divided by the fanout equals 0.33, and the node size is less than S_{disk} so the decision to expand the node is made. This is done by reallocating memory for the node, and the pointers are updated in the parent node to the reallocated address. The only change in the structure of the tree is the larger current node, shown in Figure 12. This is a key difference from the TSB-tree. We now have a current data larger than other current nodes.

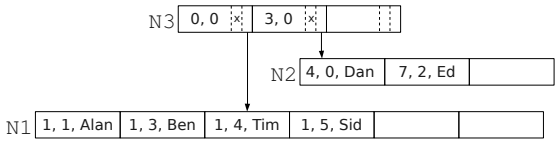


Figure 12: After a node expansion.

Time split: Time splitting nodes in the HV-tree is the same as in the TSB-tree. However, after a time split, the HV-tree tries to contract the size of the current node, and update pointers in the newly created historical node to maintain a chain of migrated nodes. Further insertions and updates are made to N1 at time 6 and 7. At time 8, key 1 needs to be updated again but N1 is full (shown in Figure 13). This time R_c is 2 for N1; R_c divided by the fanout 6 is less than T, and the node already reaches the size S_{disk}. Therefore, we time split N1. Time split is performed in a similar way as in the TSB-tree, creating a new historical node N4 and N4 is added to the migration chain. The current node N1 is contracted to the smallest allowable size that can hold the two current entries, which is S_{cache}. The the parent index entry of N4 uses the $pointer$ field to point to the current memory address of N4 and set the $hisPtr$ field to the value of O_{hist}, 0, which indicates where N4 will be placed on the hard disk when it is migrated. Then O_{hist} is incremented by S_{disk} indicating the where the next migrated node should be placed on the hard disk. Figure 14 shows the HV-tree after the time split and the values of O_{hist} and O_{mig}.

At time 9, key 5 is inserted into the node N2. At time 10, the insertion of key 8 also goes to N2 and there is not enough space. We calculate R_c divided by the fanout, which is greater than T. Therefore, we key split N2, which posts a new index entry to the root node N3. However, N3 is also full so it is also split. This is an example of splitting an index node, and the only difference is in calculating the R_c value. R_c for index nodes is determined by finding the latest time at which this node can be time split. Recall that time-splitting index nodes is restricted because we cannot have entries in historical index nodes pointing to current data nodes. The only time the N3 can be time split at is time 0. This would leave all entries remaining in the current node, which has $R_c = 3$; R_c divided by the fanout is greater than T. Therefore the decision to key split N3 (at key 3) is be made. Figure 15 shows the HV-tree after the insertion of key 8 at time 10.

Search: Suppose we want to search the HV-tree in Figure 15 for

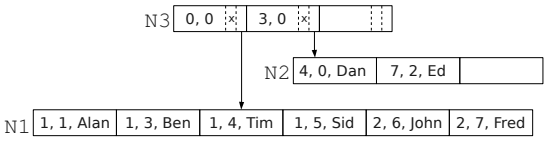


Figure 13: Before a time split.

key 1 at time 4. We start from the root, using the TSB-tree search procedure. We ignore all entries with time value beyond the search time 4 and find the entry with key-time pair such that the entry's key is not larger than the search key 1 and the entry's time is the largest time with the key 1. This leads us to N3. From there we are led to N4. In the parent index entry of N4, we find that the $histPtr$ field is 0, which is not smaller than O_{mig} (O_{mig} is 0 at the moment). Therefore, we know that N4 is still in the main memory and use the $pointer$ field to locate it. The search continues in N4. We ignore all entries with time value greater than 4, and find the latest entry with key value 1. Then we find the value associated with key 1 at time 4 was "Tim".

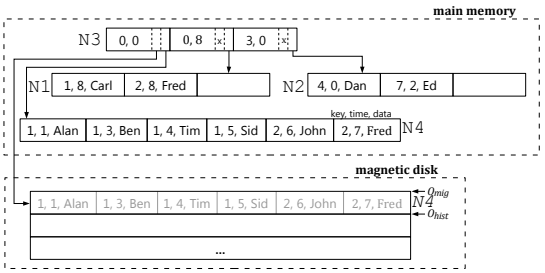


Figure 14: After a time split, node contraction, migration chain update.

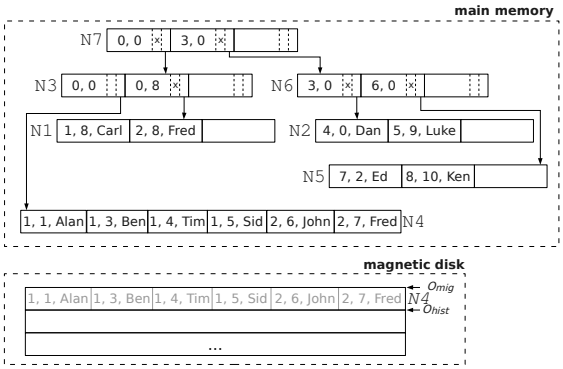


Figure 15: After a data node and index node key split.

C Node Conversion for TSB-cond

After the first node migration to disk, for any subsequent time split on a node N, we first try to find a sibling node of N which is historical and has enough space to hold the historical entries. Specifically, we scan the parent node of N to find a historical entry, and then visit the child node of this entry to determine if there is enough space to hold the historical entries. If such a node exists, we do not need to create a new historical node for the time split. The historical entries that should be put in a new historical node are now put into the sibling historical node. If we cannot find a sibling node of N with sufficient space, then we create a new historical node with S_{disk} and put the historical entries in it. Figure 16 shows an example. There is a time split at time 4 and a key split by value 4. Node N3

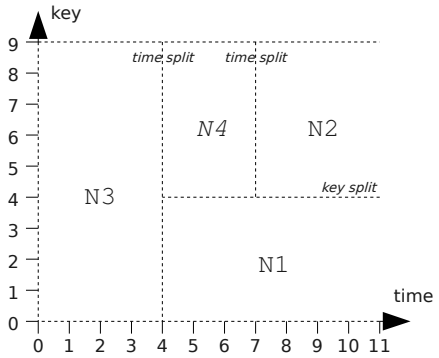


Figure 16: TSB-tree with Variable Node Sizes.

becomes a historical node, and N1 and N2 are current nodes. Later at time 7 we time split N2. In the TSB-tree, we would create a new historical node (calling it N4), but since N3 has extra space which could hold the historical data from N2, we do not create N4 (this is why it is shown in italics). Instead, N3 is used to hold the historical entries and now there are two index entries pointing to N3.

D Additional Experimental Results

D.1 Finding S_{cache}

We would like to find out the optimal node size S_{cache} for the HV-tree when all the nodes are in the main memory. Please note that this optimal node size reflects the combined effect of all the cache levels (L1, L2 and L3 if any). The main operations are updates and searches. We varied the S_{cache} value for the HV-tree from 128b to 4KB. We recorded the average update response time for a 100MB data set and subsequent average query response time. We also computed an equally weighted combination of the two, Figure 17 shows the results. We normalized the results to show just the relative performance.

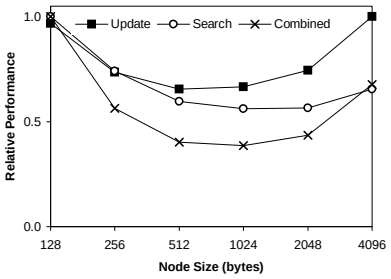
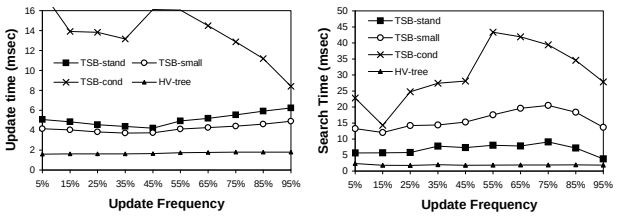


Figure 17: Tuning S_{cache} for an HV-tree.

We observe that using 512B nodes is the best for our experimental settings in terms of update performance, probably because updating involves significant amounts of data movement. Using 2KB nodes is the best for search performance. When combining updates and searches, using 1KB nodes yields the best performance. Considering a balanced workload, we have chosen 1KB as the value for S_{cache}.

D.2 Update Performance, Varying Update Frequency

In this test, we let the update frequency u vary from almost only insertions (low u values) to almost only updates (high u values). We again measure the update performance (Figure 18(a)) and the search performance (Figure 18(b)).



(a) Update (b) Search

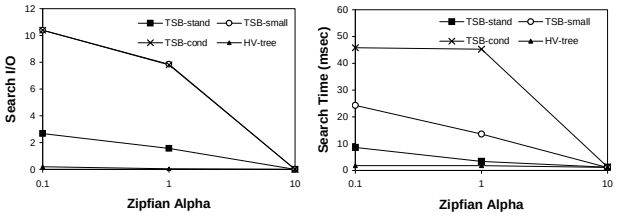
Figure 18: Varying update frequency.

We see in Figure 18(a) that for updates the HV-tree is very steady. Performing roughly the same for all update frequencies. The TSB-tree variants have fluctuating performance. TSB-small is the best performing variant, followed closely by TSB-stand. Compared to both of these, the HV-tree always requires less time to perform updates, with about 41% of the time of TSB-stand on average. Figure 18(b) shows a similar trend for the search performance. The TSB-tree variants have fluctuating performance. The HV-tree constantly performs the best, which has an average response time of only 30% that of TSB-stand.

In summary, the HV-tree has nearly constant update and search performance across all update frequencies, unlike TSB-trees which fluctuate. Also the HV-tree has much better performance than all the TSB-tree variants in all the tests.

D.3 Point Query, Varying Query Time Distribution

The query times are generated following a Zipfian distribution. In this experiment, we vary the α value of the Zipfian distribution, which controls the skewness of the data. We generated a 1,000MB dataset with queries following Zipfian distributions with the α values of 0.1, 1 and 10. The larger the α value, the more skewed the data distribution is. When α takes the large value of 10, the queries distribution is highly skewed and most of them are current queries. When α takes the small value of 0.1, the query times follow approximately a uniform distribution over the lifetime of the database, which means there are more queries on historical data than the queries with $\alpha = 10$. Figures 19(a) and Figure 19(b) show the I/O and response time of the queries.



(a) Search I/O (b) Search Time

Figure 19: Varying search query time.

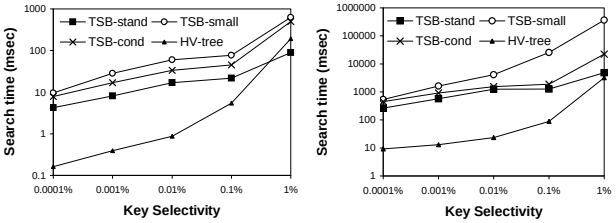
For an α value of 10, all the trees have next to zero I/Os for the queries. This is because the query time distribution is highly skewed and almost all queries are on current data. Since all the trees keep current nodes in the main memory, none of them would have any I/O when processing the queries. As the α value gets smaller, there are more queries on historical data nodes. The TSB-tree variants have notably increasing numbers of I/Os, over a magnitude of order more than the I/O of the HV-tree. The HV-tree still has near to zero I/Os even when $\alpha = 0.1$ because it keeps maximally the most recent nodes (including historical ones) in the main memory. While TSB-tree is allowed to use the operating system's buffer manager, which is an LRU, obviously it cannot optimally keep all

the most recent nodes in the main memory and therefore have increased number of I/Os when there are more queries on the historical nodes. TSB-small and TSB-cond have especially large numbers of I/Os for these cases due to their bad disk behavior. TSB-small uses small nodes and need to access the hard disk more often to retrieve the same amount of data. TSB-cond uses optimal nodes sizes for disk access, but because the data in nodes are not clustered as well as in the HV-tree, TSB-cond also has much more I/Os.

The response time comparison shown in Figure 19(b) exhibits a similar pattern to the comparison on the number of I/Os. All methods perform well when $\alpha = 10$ but the response time increases as α becomes smaller. However, the HV-tree's response time remains a very small value and outperforms all the TSB-tree variants by a large margin.

D.4 Key-Range Queries with Large Datasets

The results of key-range queries on 1,000MB datasets are shown in Figure 20. For time-slice queries (Figure 20(a)), every method



(a) Time-slice (b) Time-range

Figure 20: Key-range queries with 1000MB dataset.

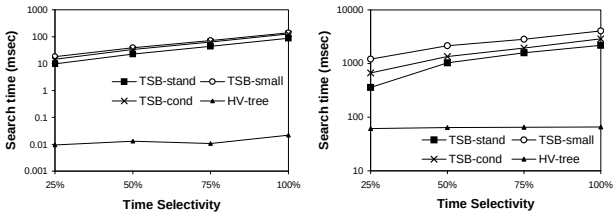
performs slower as expected. Because the datasets require all of the main memory to be used, even the HV-tree must access the hard disk. However, we still see that the HV-tree is an order of magnitude faster than the best performing TSB-tree variant, TSB-stand. As selectivity increases, the performance of HV-tree approaches that of TSB-stand. Similar to the experiments on the 500MB datasets, TSB-stand outperforms other TSB-tree variants significantly in most cases. Only at the large selectivity of 1%, the HV-tree is outperformed by TSB-stand because of the same reason given for the experiments on the 500MB datasets. Figure 20(b) shows varying key-range selectivities with a constant time-range selectivity of 50%. But this time the HV-tree is better than TSB-stand in all cases.

D.5 Time-Range Queries

In these tests we perform time-range queries. A time-range query returns a requested fraction of the history of a record. We first varied the time selectivity from 25% to 100% using the 500MB dataset. Figure 21(a) shows results when just a single key is queried and Figure 21(b) shows results when a 0.1% key-range selectivity is used.

For both Figures 21(a) and 21(b), we see that the HV-tree is far better than the TSB-tree variants. TSB-stand is still the best of the TSB-tree variants. The HV-tree is about 1000 times faster than TSB-stand. This is because the HV-tree performs less time splits (meaning less redundancy), and of course the HV-tree has better cache behavior on current nodes and more historical nodes kept in the main memory. Again, the improvement of the HV-tree over other techniques here is much higher than in update/search operations because the time-range query involves returning a lot more records than updates and point queries.

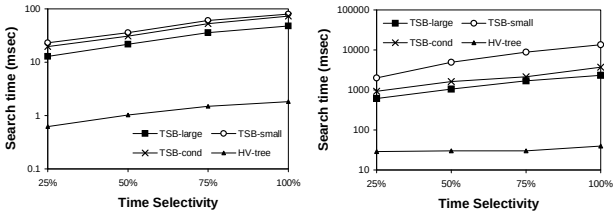
We repeated the previous tests using a 1,000MB dataset and the results are shown in Figures 22(a) and 22(b). All methods perform



(a) Single key (b) Key-range

Figure 21: Time-range queries with 500MB dataset.

slower as expected. The comparative performance is similar to that shown in the experiments using the 500MB dataset. The HV-tree outperforms all TSB-tree variants with at least an order of magnitude improvement in all tests.



(a) Single key (b) Key range

Figure 22: Time-range queries with 1000MB dataset.

In summary, the HV-tree dominates all the TSB-tree variants for time-range queries. This is because the HV-tree performs more key splits and less time splits, which means more versions of the same records are likely to exist in fewer nodes. Together with better cache behavior and delayed data migration, the HV-tree has huge advantage in time-range queries.

D.6 Verification of Threshold Value Choice

In section 4.2, we discussed our choice of T. Here we run experiments to validate the analysis. We conduct a series of tests varying the update frequency u of datasets and T values used in the implementation of the HV-tree. We average the update time, search time and memory usage over each of the update frequency values. Figure 23 shows the relative search, update and memory performance as we vary T. We also combine all three results with equal weights to give an overall "combined" performance.

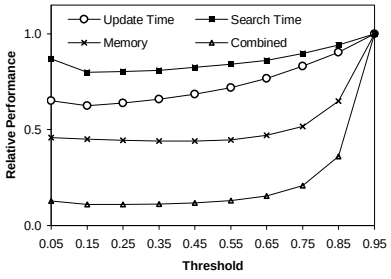


Figure 23: Investigating the choice of T.

We observe that the update and search performance is the best when T takes the values around 0.15, and $T = 0.25$ gives a very close performance to this best point. When the three measures are combined, $T = 0.25$ gives the best overall performance. This result confirms that the value of 0.25 suggested by our analysis is a very good choice. This choice of T value is quite different from the threshold value, 0.67, recommended for the TSB-tree [14, 12].